This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



# Reverse Approximate Nearest Neighbor Queries

Arif Hidayat, Shiyu Yang, Muhammad Aamir Cheema, David Taniar

**Abstract**—Given a set of facilities and a set of users, a reverse nearest neighbors (RNN) query retrieves every user $u$ for which the query facility $q$ is its closest facility. Since $q$ is the closest facility to $u$, the user $u$ is said to be influenced by $q$. In this paper, we propose a *relaxed* definition of influence where a user $u$ is said to be influenced by not only its closest facility but also every other facility that is *almost* as close to $u$ as its closest facility is. Based on this definition of influence, we propose reverse approximate nearest neighbors (RANN) queries. Formally, given a value $x > 1$, an RANN query $q$ returns every user $u$ for which $dist(u, q) \leq x \times NNDist(u)$ where $NNDist(u)$ denotes the distance between a user $u$ and its nearest facility, i.e., $q$ is an approximate nearest neighbor of $u$. In this paper, we study both *snapshot* and *continuous* versions of RANN queries. In a snapshot RANN query, the underlying data sets do not change and the results of a query are to be computed only once. In the continuous version, the users continuously change their locations and the results of RANN queries are to be continuously monitored. Based on effective pruning techniques and several non-trivial observations, we propose efficient RANN query processing algorithms for both the snapshot and continuous RANN queries. We conduct extensive experiments on both real and synthetic data sets and demonstrate that our algorithm for both snapshot and continuous queries are significantly better than the competitors.

**Index Terms**—Reverse Nearest Neighbors, Influence Detection, Continuous Monitoring, Voronoi Diagram

## 1 Introduction

People usually prefer the facilities in their vicinity, i.e., they are influenced by nearby facilities. A *reverse nearest neighbors* (RNN) query [1], [2], [3], [4] aims at finding every user that is influenced by a query facility $q$. Formally, given a set of users $U$, a set of facilities $F$ and a query facility $q$, an RNN query returns every user $u \in U$ for which the query facility $q$ is its closest facility. The set containing RNNs, denoted as $RNN(q)$, is also called the influence set of $q$.

Consider the example of Fig. 1 that shows four Starbucks cafes ($f_1$ to $f_4$) and three users ($u_1$ to $u_3$). In the context of RNN queries, the users $u_2$ and $u_3$ are both influenced by $f_1$ because $f_1$ is their closest Starbucks. Therefore, $u_2$ and $u_3$ are the RNNs of $f_1$, i.e., $RNN(f_1) = \{u_2, u_3\}$. Similarly, it can be confirmed that $RNN(f_2) = \emptyset$, $RNN(f_3) = \emptyset$, $RNN(f_4) = \{u_1\}$.

A *reverse $k$ nearest neighbors* (R$k$NN) query [5], [6], [7], [8], [9], [10] is a natural extension of the RNN query and uses a *relaxed* notion of influence. Specifically, in the context of an R$k$NN query, a user $u$ is considered to be influenced by its $k$ closest facilities. Hence, an R$k$NN query $q$ returns every user $u \in U$ for which $q$ is among its $k$ closest facilities. In the example of Fig. 1, assuming $k = 2$, $R2NN(f_2) = \{u_1, u_2, u_3\}$ because $f_2$ is one of the two closest facilities for all of the three users. Similarly, $R2NN(f_1) = \{u_2, u_3\}$, $R2NN(f_3) = \emptyset$ and $R2NN(f_4) = \{u_1\}$.

R$k$NN queries have numerous applications [1] in location based services, resource allocation, profile-based management, decision support etc. Consider the example of a supermarket. The people for which this supermarket is one of the $k$ closest supermarkets are its potential customers and may be influenced by targeted marketing or special deals. Due to its significance, RNN queries and its variants have received significant research attention in the past decade (see [6] for a survey).

In this paper, we propose an alternative definition of influence and propose a variant of RNN queries called *reverse approximate nearest neighbors* (RANN) query. This definition is motivated by our observation that an R$k$NN query may not properly capture the notion of influence.

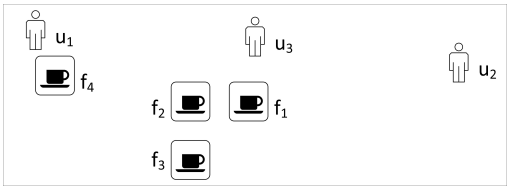



Fig. 1: Illustration of RNN query and its variants

### 1.1 Motivation

Consider the example of a person living in a suburban area (e.g., $u_2$ in Fig. 1) who does not have any Starbucks nearby. Her nearest Starbucks is $f_1$ which is say 30 Km from her location. In the context of R2NN query, $u_2$ is influenced by $f_1$ and $f_2$ – her two nearest facilities. However, we argue that it is also influenced by $f_3$ because a user who needs to travel a minimum of 30 Km to visit a Starbucks may also be willing to travel to a Starbucks cafe 31 Km far from her.

Similarly, consider the example of another person living in a suburb (e.g., $u_1$ in Fig. 1) who has only one Starbucks nearby ($f_4$) assuming that all other Starbucks (e.g., $f_1$ to $f_3$) are in downtown area and are quite far. In the context of R2NN queries, the user $u_1$ is considered to be influenced by both $f_4$ and $f_2$ because these are her two closest facilities. However, we argue that the user $u_1$ is only influenced by


- *A. Hidayat is with the Faculty of Information Technology, Monash University, Australia and with Brawijaya University, Indonesia. E-mail: arif.hidayat@monash.edu*
- *S. Yang is the corresponding author. He is with the School of Computer Science and Engineering, The University of New South Wales, Australia E-mail: yangs@cse.unsw.edu.au*
- *M. A. Cheema and D. Taniar are with the Faculty of Information Technology, Monash University, Australia E-mail: {aamir.cheema, david.taniar}@monash.edu*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



$f_4$ because the other facilities are significantly farther than $dist(u_1, f_4)$, e.g., a user who has a Starbucks within 1 Km is not very likely to visit a Starbucks that is say 30 Km from her location.

As shown above, the definition of influence used in R$k$NN queries considers only the relative ordering of the facilities based on their distances from $u$ and ignores the actual distances of the facilities from $u$. Motivated by this, in this paper, we propose a reverse approximate nearest neighbors (RANN) query that relaxes the definition of influence using a parameter $x$ (called the $x$ factor in this paper) and considers the relative distances between users and facilities. Specifically, an RANN query returns every user $u$ for which the query facility is its approximate nearest neighbor.

**Definition 1.** ***Approximate nearest neighbor****. Let $NNdist(u)$ denote the distance between $u$ and its nearest facility. Given a value of $x > 1$, a facility $f$ is called an approximate nearest neighbor of $u$ if $dist(u, f) \leq x \times NNdist(u)$.*

**Reverse Approximate Nearest Neighbors (RANN) query**. Given a value of $x > 1$, an RANN query $q$ returns every user $u$ for which $dist(u, q) \leq x \times NNdist(u)$, i.e., return every user $u$ for which $q$ is its approximate nearest neighbor. The set of RANNs of a query $q$ is denoted as $RANN_x(q)$. Note that an RANN query is the same as an RNN query if $x = 1$.

In the example of Fig. 1, assuming $x = 1.2$, RANN of $f_2$ are the users $u_2$ and $u_3$, i.e., $RANN_{1.2}(f_2) = \{u_2, u_3\}$. Similarly, $RANN_{1.2}(f_1) = \{u_2, u_3\}$, $RANN_{1.2}(f_3) = \{u_2\}$ and $RANN_{1.2}(f_4) = \{u_1\}$.

In this paper, we study both *snapshot* and *continuous* RANN queries. In a snapshot RANN query, the users and facilities do not change their locations and the results of a query are required to be computed only once. In a continuous RANN query, the facilities (e.g., fuel stations) do not change their locations but the users (e.g., drivers) are continuously moving. In such scenario, the results are to be continuously monitored and reported to the query facility. For instance, a fuel station owner may want to continuously monitor the cars influenced by it and may send them promotions. Similarly, a restaurant owner may want to continuously monitor the near by customers who may be influenced by targeted deals.

*Remark.* R$k$NN queries and RANN queries assume that the distance is the main factor influencing a user. This assumption holds in many real world scenarios. For instance, the users looking for nearby fuel stations are usually not concerned about price (or even rating) because all fuel stations have similar price (or even the same price because, in some countries, the fuel prices are regulated by the government). Similarly, users interested in McDonald's restaurants or Starbucks cafe are mainly influenced by the distance because other attributes such as price, menu, and ratings are the same for all stores. Nevertheless, in the case where the users are influenced by other attributes, reverse top-$k$ queries [11], [12], [13] can be used to compute the influence using a scoring function involving multiple attributes such as distance, price, and rating. This is a different line of research and is not within the scope of this paper.

### 1.2 Contributions

We make the following contributions in this paper.

**1.** We complement the R$k$NN queries by proposing a new definition of influence that considers every user $u$ to be influenced by a query $q$ for which $q$ is an approximate nearest neighbor.

**2.** As we show in Section 3.2, the pruning techniques used to solve R$k$NN queries cannot be applied or extended for RANN queries. This is mainly because, in our problem settings, a facility $f$ may not be able to prune the users that are quite far from $f$ (see Section 3.2 for details). Based on several non-trivial observations, we propose efficient pruning techniques that are proven to be *tight*, i.e., given a facility $f$ used for pruning, the pruning techniques guarantee to prune every point that can be pruned by $f$. We then propose an efficient algorithm that utilizes these pruning techniques to efficiently answer the snapshot RANN queries.

This paper is an extended version of our earlier work [14]. We make the following new contributions.

**3.** The existing techniques for RANN queries can only be applied for snapshot RANN queries and their extension for continuous RANN queries is not trivial. Using the snapshot RANN algorithm for every timestamp to monitor continuous RANN queries is prohibitively expensive. Therefore, we propose a novel Voronoi-based algorithm to efficiently monitor concurrent continuous RANN queries for moving objects. A by-product of our techniques for continuous RANN queries is an alternative Voronoi-based algorithm to solve snapshot RANN queries. This Voronoi-based algorithm outperforms our previous algorithm [14] by up to 20 times. To be fair with our previous algorithm, the new Voronoi-based algorithm requires a pre-computed Voronoi diagram and some changes to the standard indexes (e.g., R*-tree, Quadtree). In contrast, our previous algorithm (Section 3.2) can be applied on any branch-and-bound index without requiring any modification. Therefore, the previous algorithm may be preferred by a system administrator who does not want to modify the existing indexes.

**4.** We conduct an extensive experimental study on both real and synthetic data sets to show the effectiveness of our proposed techniques. Since existing techniques cannot be extended to answer snapshot RANN queries, we compare our snapshot algorithm with a significantly improved version of a naïve algorithm called improved range query (IRQ). The experimental results show that both of our snapshot algorithms are several orders of magnitude better than the competitor. The newly proposed Voronoi-based algorithm significantly outperforms our other algorithm.

For continuous RANN queries, we present a non-trivial extension of the state-of-the-art R$k$NN monitoring algorithm [8] to handle RANN queries and use it as a competitor for our Voronoi-based algorithm. Our experiments show that our proposed algorithm significantly outperforms the state-of-the-art algorithm in terms of both initial computation cost and continuous monitoring cost

The rest of the paper is organized as follows. We present an overview of the related work in Section 2. The problem definition, pruning techniques and algorithm for snapshot RANN queries are discussed in Section 3. Section 4 presents the proposed solution for continuous monitoring of RANN

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



queries. An extensive experimental study is provided in Section 5 followed by conclusions in Section 6.

## 2 Related Work

The R$k$NN query has been extensively studied [2], [3], [4], [5], [7], [8], [9], [10], [15], [16], [17], [18], [19], [20] ever since it was introduced in [1]. There has also been significant interest in other variants of R$k$NN queries such as reverse top-$k$ queries [11], [12], spatial reverse top-$k$ queries [13], location constraint queries [21], reverse spatial keyword queries [22], reverse furthest neighbors [23] and reverse skyline queries [24], to name a few. Due to the space limitations, in this section we focus only on R$k$NN queries.

### 2.1 Snapshot RNN Queries

We briefly describe two widely used pruning strategies.

**Half-space based pruning [5]**. A perpendicular bisector between a facility $f$ and a query $q$ divides the whole space into two halves. Let $H_{f:q}$ denote the half-space that contains $f$ and $H_{q:f}$ be the half-space that contains $q$. A user $u$ that lies in $H_{f:q}$ cannot be the RNN of $q$ because $dist(u, f) < dist(u, q)$. Consider the example of Fig. 2, where the half-space $H_{a:q}$ is the shaded area. The users $u_1$ and $u_2$ cannot be the RNN of $q$ because they lie in $H_{a:q}$. This observation can be extended for R$k$NN queries. Specifically, a user $u$ cannot be the R$k$NN of $q$ if it lies in at least $k$ such half-spaces. In Fig. 2, assuming $k = 2$, the user $u_2$ cannot be R2NN of $q$ because it lies in $H_{a:q}$ and $H_{b:q}$. In other words, the area $H_{a:q} \cap H_{b:q}$ (the dark shaded area) can be pruned.

**Six-regions based pruning [2].** In six-regions based pruning approach, the space around $q$ is divided into six equal regions of $60°$ each (see $P_1$ to $P_6$ in Fig. 3). Let $d_i^k$ be the distance between $q$ and its $k$-th nearest facility in a partition $P_i$. It can be proved that a user $u$ lying in a partition $P_i$ cannot be the R$k$NN of $q$ if $dist(u, q) > d_i^k$. Based on this observation, the $k$-th nearest facility in each partition $P_i$ is found and the distance $d_i^k$ is used to prune the search space. For instance, in Fig. 3, the shaded area can be pruned if $k = 1$, i.e., the users $u_1$ and $u_2$ are pruned.

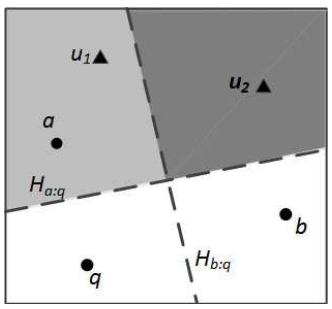



Fig. 2: Half-space pruning

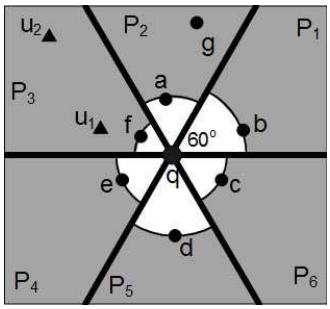



Fig. 3: Six-regions pruning

It has been shown [5] that the half-space based approach prunes more area than the six-regions based pruning. However, the advantage of the six-regions based pruning is that it is computationally less expensive. Six-region [2] and SLICE [10] are the most notable algorithms that use six-regions based pruning whereas TPL [5], FINCH [25], InfZone [8], [26], and TPL++ [6] are some of the remarkable algorithms that employ half-space based pruning. The details of these algorithms can be found in a recent survey paper [6].

To the best of our knowledge, none of the existing algorithms can be applied or trivially extended to answer RANN queries studied in this paper.

### 2.2 Continuous RNN Queries

The first algorithm for continuous monitoring of RNN queries was presented in [27]. In this work, objects' velocities are assumed to be known. Xia *et al.* [28] proposed the first algorithm that does not assume any knowledge about the objects' motion pattern. This algorithm works based on six region approach and monitors the RNN queries by monitoring the unpruned area. Kang *et al.* [29] proposed RNN monitoring algorithm based on half-space (TPL) pruning approach that also monitors the unpruned area.

The first R$k$NN monitoring algorithm was proposed by Wu *et al.* in [25]. In this algorithm, $k$NN queries are issued in each region and the users that are closer than the $k$-th NN become the candidates and are verified if the query point is one of their $k$ closest facilities. The R$k$NN of queries are monitored using the circle that contains $k$ nearest facilities for each candidate object. Cheema *et al.* [19] proposed an algorithm, called Lazy Updates, that reduces the number of times pruning phase is executed. In this algorithm, they assign a safe region for each moving object such that if the object stays in the region, the pruning phase is not needed to be called.

Cheema *et al.* [8], [26] present the *state-of-the-art* algorithm for continuous R$k$NN queries. They introduce the notion of *influence zone* of a query $q$ which is an area such that a user $u$ is a R$k$NN of the query $q$ if and only if $u$ is inside this area. Thus, once the influence zone of a query is constructed, the system can efficiently monitor the R$k$NNs of a query by monitoring the users that enter or leave its influence zone.

To efficiently monitor the users that enter or leave the influence zone, the space is partitioned using a grid containing $N \times N$ cells. A cell is called an *interior* cell if it is fully contained by the influence zone. A cell is called a *border* cell if it is partially contained by the influence zone. Consider polygon *ABCDEFGHI* in Fig. 4 which is the influence zone of $q$. The dark shaded cells are the *interior* cells and the light shaded ones are the *border* cells. For each cell, the algorithm maintains two lists called *interior list* and *border list*. The interior (resp. border) list of a cell $c$ consists of every query $q$ for which the cell $c$ is an interior cell (resp. a border cell). Whenever a user $u$ enters a cell $c$, it becomes the R$k$NNs of every query in the interior list of $c$. The algorithm also checks every query in the border list of $c$ and checks whether $u$ is inside its influence zone or not and updates the results accordingly.

## 3 Snapshot RANN Queries

### 3.1 Problem Definition

Given a set of users $U$, a set of facilities $F$, a query facility $q$ (which may or may not be in $F$), and a value of $x > 1$, a reverse approximate nearest neighbors (RANN) query returns

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



every user $u \in U$ for which $dist(u,q) \leq x \times NNdist(u)$ where $NNDist(u)$ denotes the distance between $u$ and its nearest facility in $F$.

Although our techniques can be applied on any branch-and-bound data structure, in this paper, we assume that both the facility and user data sets are indexed by R*-tree [30]. The R*-tree that indexes the set of facilities (resp. users) is called facility (resp. user) R*-tree. Since most of the applications of the RNN query and its variants are in location-based services, similar to the existing RNN algorithms [6], the focus of this paper is on two dimensional location data.

### 3.2 Pruning Techniques

Given a facility $f$, a user $u$ cannot be the RANN of $q$ if $dist(u,q) > x \times dist(u,f)$. In such case, we say that the facility $f$ prunes the user $u$. In this section, we will present the pruning techniques that use a facility $f$ or an MBR of the facility R*-tree to prune the users. First, we highlight the challenges.

#### 3.2.1 Challenges

Existing pruning techniques for R$k$NN queries cannot be applied or extended for the RANN queries regardless of the value of $k$ chosen. We illustrate this using an example. Assume that all the facilities in the data set $F$ are clustered together. Consider a user $u$ that is quite far from all these facilities (e.g., the closest facility is 100km away and the furthest facility is 101 km away). Now assume that the query facility is 102 km away from the user $u$. Clearly, $dist(u,f) < dist(u,q)$ for every $f \in F$. In other words, $u$ cannot be R$k$NN of $q$ for any value of $k$ between 1 to $|F|$, i.e., $u$ is pruned by all $|F|$ facilities as per the definition of R$k$NN queries. On the other hand, assume that $x = 1.05$. The user $u$ is a reverse approximate nearest neighbor of $q$ because $q$ is its approximate nearest neighbor, i.e. $dist(u,q) < x \times dist(u,f')$ where $f'$ is the nearest facility of $u$. Note that the above example does not assume any particular pruning approach and holds for every possible pruning approach (whether existing or yet to be developed) for R$k$NN queries.

The pruning for RANN queries is more challenging than the pruning for R$k$NN queries. For instance, the algorithms to solve R$k$NN queries can prune most of the search space by considering *only* the nearby facilities surrounding $q$. Consider the example of Fig. 3 where the six-regions approach finds the nearest facility to the query $q$ in each of the six partitions and the shaded area can be pruned. For example, the users $u_1$ and $u_2$ in the partition $P_3$ can be pruned by the facility $f$.

However, in the case of RANN queries, the nearby facilities surrounding the query $q$ are not sufficient to prune a large part of the search space. Assuming $x = 2$, while the user $u_1$ can be pruned by $f$ the user $u_2$ cannot be pruned by $f$ (see Fig. 5 where the pruned area is shown shaded). In other words, the users that are further from a facility $f$ are less likely to be pruned by it.

In Fig. 5, assuming $x = 2$, the six shaded circles show the maximum possible area that can be pruned by the six facilities $a$ to $f$ (the details on how to compute the circles will be presented later). Note that the facilities that are close to $q$ prune a smaller area as compared to the farther facilities. Hence, the algorithm needs to access not only nearby facilities but also farther facilities to prune a large part of the search space. Also, note that RANN queries are more challenging because the maximum area that can be pruned is significantly smaller.

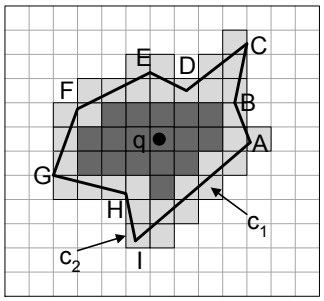



Fig. 4: R$k$NN monitoring [8]

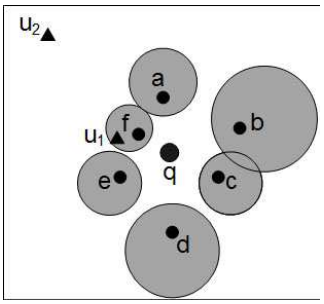



Fig. 5: Challenges

In Section 3.2.2, we present the pruning techniques that prune the space using a data point, i.e. a facility $f$. In Section 3.2.3, we present the techniques to prune the space using an MBR of the facility R*-tree.

#### 3.2.2 Pruning using a facility point

Before we present our non-trivial pruning technique, we present the definition of a pruning circle.

**Definition 2** (Pruning circle)**.** *Given a query $q$, a multiplication factor $x > 1$ and a point $p$, the pruning circle of $p$ (denoted as $C_p$) is a circle centered at $c$ with radius $r$ where $r = \frac{x \cdot dist(q,p)}{x^2-1}$ and $c$ is on the line passing through $q$ and $p$ such that $dist(q,c) > dist(p,c)$ and $dist(q,c) = \frac{x^2 \cdot dist(q,p)}{x^2-1}$.*

Consider the example of Fig. 6 that shows the pruning circle $C_f$ of a facility $f$ assuming $x = 2$. The centre of $c$ is located on the line passing through $q$ and $f$ such that $dist(q,c) = \frac{4 \cdot dist(q,f)}{3}$, $dist(q,c) > dist(f,c)$ and radius $r = \frac{2 \cdot dist(q,f)}{3}$. The condition $dist(q,c) > dist(f,c)$ ensures that $c$ lies towards $f$ on the line passing through $q$ and $f$, i.e., $f$ lies between the points $c$ and $q$ as shown in Fig. 6. Next, we introduce our first pruning rule in Lemma 1.

**Lemma 1.** *Every user $u$ that lies in the pruning circle $C_f$ of a facility $f$ cannot be the RANN of $q$, i.e., $dist(u,q) > x \times dist(u,f)$.*

*Proof.* Given two points $v$ and $w$, we use $\overline{vw}$ to denote $dist(v,w)$. Consider the example of Fig. 6. Since $u$ is inside the circle $C_f$, $\overline{uc} < r$. Assume that $\overline{uc} = n \cdot r$ where $0 \leq n < 1$. Since $r = \frac{x \cdot \overline{qf}}{x^2-1}$, we have $\overline{uc} = n \cdot r = n \cdot \frac{x \cdot \overline{qf}}{x^2-1}$.

Considering the triangle $\triangle quc$, $\overline{qu} = \sqrt{(\overline{qc})^2 + (\overline{uc})^2 - 2 \cdot \overline{uc} \cdot \overline{qc} \cdot \cos\theta}$. Since $\overline{uc} = n \cdot \frac{x \cdot \overline{qf}}{x^2-1}$ and $\overline{qc} = \frac{x^2 \cdot \overline{qf}}{x^2-1}$, we have

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



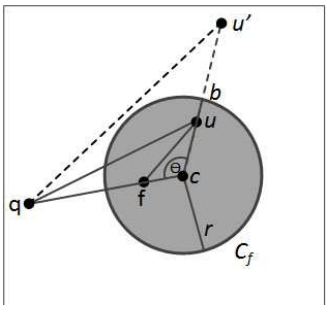



Fig. 6: Lemma 1

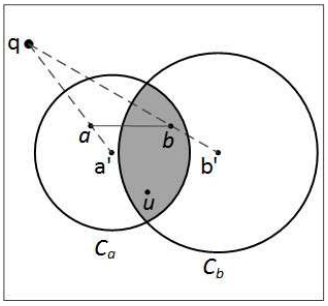



Fig. 7: Lemma 3

$$\overline{qu} = \Big[(\frac{x^2 \cdot \overline{qf}}{x^2-1})^2 + n^2(\frac{x \cdot \overline{qf}}{x^2-1})^2 - 2n(\frac{x \cdot \overline{qf}}{x^2-1})(\frac{x^2 \cdot \overline{qf}}{x^2-1}) \cdot \cos\theta\Big]^{\frac{1}{2}}$$
$$= \sqrt{(\frac{x \cdot \overline{qf}}{x^2-1})^2(x^2 + n^2 - 2 \cdot x \cdot n \cdot \cos\theta)}$$
$$= (\frac{x \cdot \overline{qf}}{x^2-1})\sqrt{x^2 + n^2 - 2xn\cos\theta} \quad (1)$$

Similarly considering $\triangle fcu$, $\overline{fu} = \sqrt{(\overline{fc})^2 + (\overline{uc})^2 - 2 \cdot \overline{uc} \cdot \overline{fc} \cdot \cos\theta}$. Since $\overline{fc} = \overline{qc} - \overline{qf}$ and $qc = \frac{x^2 \cdot \overline{qf}}{x^2-1}$, we get $\overline{fc} = \frac{\overline{qf}}{x^2-1}$. We can obtain the value of $\overline{fu}$ by replacing the values of $\overline{fc}$ and $\overline{uc}$.

$$\overline{fu} = \Big[(\frac{\overline{qf}}{x^2-1})^2 + n^2(\frac{x \cdot \overline{qf}}{x^2-1})^2 - 2 \cdot n(\frac{x \cdot \overline{qf}}{x^2-1}) \cdot (\frac{\overline{qf}}{x^2-1}) \cdot \cos\theta\Big]^{\frac{1}{2}}$$
$$= (\frac{\overline{qf}}{x^2-1})\sqrt{1 + n^2x^2 - 2nx\cos\theta} \quad (2)$$

Note that the user $u$ can be pruned if $dist(u,q) > x \times dist(u,f)$. Therefore, we need to show $\overline{qu} - x \cdot \overline{fu} > 0$. The left side of this inequality can be obtained using the values of $\overline{qu}$ and $\overline{fu}$ from Eq. (1) and Eq. (2), respectively.

$$\overline{qu} - x \cdot \overline{fu} = (\frac{x \cdot \overline{qf}}{x^2-1})\sqrt{x^2 + n^2 - 2.x.n.cos(\theta)} - x. (\frac{\overline{qf}}{x^2-1})\sqrt{x^2.n^2 + 1 - 2.x.n.cos(\theta)}$$
$$= \frac{x.\overline{qf}}{x^2-1}\Big(\sqrt{x^2 + n^2 - 2xn\cos\theta} - \sqrt{1 + x^2n^2 - 2xn\cos\theta}\Big) \quad (3)$$

Since $x > 1$, $(\frac{x.\overline{qf}}{x^2-1})$ is always positive. Hence, we just need to prove that $(\sqrt{x^2 + n^2 - 2xn\cos\theta} - \sqrt{1 + x^2n^2 - 2xn\cos\theta} > 0$. In other words, we need to show $(\sqrt{x^2 + n^2 - 2xn\cos\theta} > \sqrt{1 + x^2n^2 - 2xn\cos\theta}$. Note that both sides of this inequality are positive (otherwise $\overline{qu}$ and $\overline{fu}$ in Eq. (1) and Eq. (2) would be negative which is not possible). Hence, we can take the square of both sides resulting in $x^2 + n^2 - 2xn\cos\theta > 1 + x^2n^2 - 2xn\cos\theta$ which implies that we need to prove $(x^2 + n^2 - x^2n^2 - 1) > 0$. This inequality can be simplified as $(x^2-1)(1-n^2) > 0$. Since $x > 1$ and $n < 1$, it is easy to see that $(x^2-1)(1-n^2) > 0$ which completes the proof. □

Note that although the pruning technique itself is non-trivial, applying this pruning rule is not expensive, i.e., to check whether a user $u$ can be pruned or not, we only need to compute its distance from the centre $c$ and compare it with the radius $r$. Next, we show that this pruning rule is *tight* in the sense that any user $u'$ that lies outside $C_f$ is guaranteed not to be pruned by the facility $f$.

**Lemma 2.** *Given a facility $f$ and a user $u'$ that lies on or outside its pruning circle $C_f$, then $dist(u',q) \leq x \times dist(u',f)$, i.e. $u'$ cannot be pruned by $f$.*

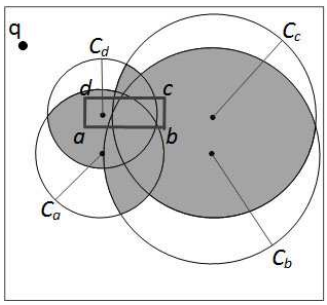
*Proof.* Consider the user $u'$ in Fig. 6. Since $u'$ is on or outside the pruning circle, it satisfies $\overline{u'c} = n \cdot r$, where $n \geq 1$. The proof is similar to the proof of Lemma 1 except that we need to show that $\overline{u'q} - x.\overline{fu'} \leq 0$, i.e., we need to show $(x^2-1)(1-n^2) \leq 0$ which is obvious given that $x > 1$ and $n \geq 1$. □

Note that the pruning circle $C_f$ is larger if $dist(q,f)$ is larger which implies that the facilities that are farther from the query prune larger area. For instance, in Fig. 7, the pruning circle $C_b$ is bigger than the pruning circle $C_a$.

### 3.2.3 Pruning using the nodes of facility R*-tree

In this section, we present our techniques to prune the search space using the intermediate or leaf nodes of the facility R*-tree. These pruning techniques reduce the I/O cost of the algorithm because the algorithm may prune the search space using a node of the R*-tree instead of accessing the facilities in its sub-tree.

A node of the facility R*-tree is represented by a minimum bounding rectangle (MBR) that encloses all the facilities in its sub-tree. Without accessing the contents of the node, we cannot know the locations of the facilities inside it except that each side of the MBR contains at least one facility. We utilize this information to devise our pruning techniques. Specifically, we use all four sides of the MBR and use each side (i.e., line segment) to prune the search space. Lemma 3 presents the pruning rule and Fig. 7 provides an illustration.



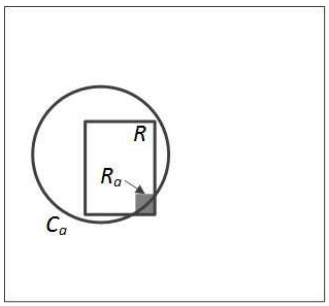




Fig. 8: Pruning using MBR  Fig. 9: Trimming an MBR

**Lemma 3.** *Given a query $q$, a multiplication factor $x > 1$, and a line $\overline{ab}$ representing a side of an MBR, a user $u$ cannot be the*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



*RANN of q if it lies inside both of the pruning circles $C_a$ and $C_b$, i.e., u can be pruned if u lies in $C_a \cap C_b$.*

*Proof.* Let $maxdist(p, \overline{ab})$ denote the maximum distance between a point $p$ and a line $\overline{ab}$. Note that $maxdist(u, \overline{ab}) = max(dist(u, a), dist(u, b))$. Since $u$ lies in both $C_a$ and $C_b$, $dist(u, q) > x \times dist(u, a)$ and $dist(u, q) > x \times dist(u, b)$ (according to Lemma 1). In other words, $dist(u, q) > x \times maxdist(u, \overline{ab})$. Since there is at least one facility $f$ on the line $\overline{ab}$, $dist(u, f) \leq maxdist(u, \overline{ab})$. Hence, $dist(u, q) > x \times dist(u, f)$ which implies that the user $u$ can be pruned. □

In Fig. 7, the shaded area can be pruned by using the line $\overline{ab}$. The next lemma shows that this pruning rule is also tight.

**Lemma 4.** *Given a line $\overline{ab}$ such that the only information we have is that there is at least one facility $f$ on $\overline{ab}$, a user $u$ cannot be pruned if it lies outside either $C_a$ or $C_b$.*

*Proof.* Without the loss of generality, assume that $u$ lies outside $C_a$. Now assume that there is exactly one facility $f$ on the line $\overline{ab}$ and it lies at the end point $a$. Since $f$ lies on $a$, $C_a = C_f$ which implies that $u$ is outside $C_f$. Hence, $u$ cannot be pruned by $f$ (Lemma 2). □

To prune the search space using an MBR, we apply Lemma 3 on each of side $s_i$ of the MBR. Specifically, a user $u$ can be pruned if, for *any* side $s_i$ of the MBR, $u$ lies in *both* of the pruning circles of the end points of $s_i$. Consider the example of Fig. 8 where an MBR $abcd$ is shown along with the pruning circles of the corners of the MBR (see $C_a$ to $C_d$). Let $A_i$ denote the area pruned by a side $s_i$ of the MBR. In Fig. 8, the shaded area can be pruned which corresponds to $\cup_{i=1}^{4} A_i$ where $A_1 = C_a \cap C_b$, $A_2 = C_b \cap C_c$, $A_3 = C_c \cap C_d$, and $A_4 = C_d \cap C_a$.

The efficient implementation of the pruning techniques presented in this section are discussed in the conference version [14] of this paper.

### 3.3 Algorithm

Our algorithm consists of three phases namely *pruning*, *filtering* and *verification*. In the pruning phase, we traverse the facility R*-tree to shortlist facility MBRs and facility points to prune the search space. In the filtering phase, the users that lie in the pruned space are pruned and the remaining users are inserted in a candidate list called $L_{cnd}$. Finally, in the verification phase, each candidate user $u \in L_{cnd}$ is verified to check whether it is an RANN of $q$ or not.

**Pruning Phase**. In this pase, we traverse the facility R*-tree to shortlist facility entries (i.e., MBRs and points) to prune the search space. An *aggressive* pruning approach is to shortlist every facility entry that prunes at least one point in the search space not pruned by the previously considered facility entries. This approach ensures that the pruned space is maximized, i.e., any user $u$ that lies outside the pruned space is guaranteed to be an answer. However, our experimental study shows that this significantly increases the pruning and filtering cost because the number of facility entries considered for pruning and filtering may be quite high. Therefore, similar to most of the existing work on R$k$NN queries (e.g., see a survey in [6]), we use a *moderate* pruning approach which shortlists only the facility entries that lie outside the currently pruned search space. This is inspired by the observation that a facility entry that lies inside the pruned area is less likely to prune additional area. Although the space pruned by the moderate approach is not maximized, this significantly reduces the pruning and filtering cost which results in a significant improvement on overall query processing cost. The experimental results comparing the aggressive and moderate pruning techniques are presented in Section 5.1.

Algorithm 1 presents the details of our moderate pruning technique. The algorithm initializes a heap $h$ with the root of the facility R*-tree. The entries are iteratively deheaped from the heap and are processed as follows. If a deheaped entry $e$ lies in the pruned space, we ignore it (lines 5 and 6). Otherwise, we process it as follows.

**Algorithm 1 Pruning**

**Input:** facility R*-tree, and a query $q$
**Output:** The set of pruned areas $\mathcal{A}$

```
1:  A ← φ
2:  insert root of facility R-tree in a h
3:  while h is not empty do
4:      de-heap an entry e
5:      e' ← PruneEntry(e, A)              ▷ Algorithm 1 in [14]
6:      if e' ≠ φ then                     ▷ e is not pruned
7:          if e is an intermediate or leaf node then
8:              for each side ab of e do
9:                  create A_i = C_a ∩ C_b and insert in A
10:             for each child c of e do
11:                 if c overlaps with e' then insert c in the heap
12:         else                           ▷ e is a facility point
13:             create A_i = C_e and insert in A
```

If $e$ is an intermediate or leaf node of the R*-tree, for each side of $e$, we create a pruning area $A_i$ and insert it in $\mathcal{A}$ (line 9). We also insert its children in the heap $h$. Note that a child $c$ of $e$ that does not overlap with $e'$ can be pruned because it lies in the pruned area. Hence, only the children that overlap with $e'$ are inserted in the heap (line 11). If $e$ is a facility point, we create the pruning circle $C_e$ and add it to $\mathcal{A}$ (line 13). The algorithm terminates when the heap becomes empty.

**Filtering Phase**. Algorithm 2 describes the filtering phase. A stack $S$ is initialized with the root entry of the user R*-tree. Each entry $e$ is iteratively retrieved from $S$ and processed as follows. If $e$ can be pruned by $\mathcal{A}$, it is ignored (lines 5 and 6). Otherwise, if it is an intermediate or leaf node, its children that overlap with $e'$ are inserted in the stack (line 9). If $e$ is a user, it is inserted in $L_{cnd}$ (line 11). The algorithm stops when the stack $S$ becomes empty.

**Verification Phase**. In the verification phase, each candidate user $u \in L_{cnd}$ is verified as follows. Note that a user $u$ is an RANN if and only if there is no facility $f$ for which $dist(u, f) < \frac{dist(u,q)}{x}$. A circular boolean range query is issued with centre at $u$ and radius $r = \frac{dist(u,q)}{x}$ that returns true if and only if there exists a facility in the circle. The boolean range query is conducted on the facility R*-tree as in previous works [7] and $u$ is reported as an answer if it returns false.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



**Algorithm 2 Filtering**

**Input:** user R*-tree, query $q$, and $\mathcal{A}$
**Output:** a list of candidates $L_{cnd}$

```
1: L_cnd ← φ
2: insert root of user R*-tree in a stack S
3: while S is not empty do
4:    retrieve top entry e from S
5:    e' ← PruneEntry(e, A)            ▷ Algorithm 1 in [14]
6:    if e' ≠ φ then                   ▷ e is not pruned
7:       if e is an intermediate or leaf node then
8:          for each child c of e do
9:             if c overlaps with e' then insert c in stack S
10:      else                          ▷ e is a user
11:         insert e in L_cnd
```

## 4 Continuously Monitoring RANN Queries

In this section, we present techniques to continuously monitor RANN queries. Given a set of facilities $F$, a set of users $U$, a set of queries $Q$ and a value of $x > 1$, the problem of continuous monitoring of RANN queries is to continuously monitor the RANNs of every $q \in Q$ when one or more users change their locations.

### 4.1 System Overview

We assume a client-server paradigm. The server maintains the locations of facilities and the moving users. When a client issues a query, the server computes and sends its initial results to the client. The server also assigns each moving user a *safe zone* which is an area such that the user's movement within this area does not affect the results of any query in the system. The user reports its location to the server only when it leaves its respective safe zone. In this case, the server updates the results of affected queries and sends the relevant clients the updated results. Then, the server computes the new safe zone and sends it to the user. The system also maintains up-to-date results when queries and/or users are added to or deleted from the system. Like most of the existing work on continuous queries [19], we assume a timestamp model in which the server receives the location updates at each timestamp (e.g., after every $t$ time units) and updates the results accordingly.

### 4.2 Solution Overview

Recall that a user $u$ is an RANN of a query $q$ if and only if it lies outside the pruning circle of every facility (Lemma 1 and Lemma 2). Thus, a straightforward approach to verify whether a user $u$ is an RANN of a query $q$ is to check pruning circles of all facilities with respect to $q$ and determine if $u$ is outside all these circles or not. Consider a query facility $q$ and two other facilities $f_1$ and $f_2$ in Fig. 10. $C_{f_1}$ (resp. $C_{f_2}$) is the pruning circle of facility $f_1$ (resp. $f_2$) with respect to the query $q$. In this example, $u_2$ is an RANN of $q$ as it is outside all pruning circles. On the other hand, $u_1$ is not an RANN of $q$ since there is at least one pruning circle ($C_{f_2}$) that contains it. This simple approach requires $O(|F|)$ to check whether a given user $u$ is an RANN of a query $q$ where $|F|$ is the total number of facilities.

Next, we present an observation that allows checking whether a user is an RANN of a query or not by considering only one pruning circle.

**Lemma 5.** *Let $f$ be the nearest facility of a user $u$. $u$ is an RANN of a query $q$ if and only if $u$ is outside the pruning circle $C_f$ of $f$.*

*Proof.* If $u$ is inside the pruning circle $C_f$, it cannot be an RANN of $q$ (Lemma 1), e.g $u_1$ in Fig. 10. Next, we show that if $u$ is outside of $C_f$, it is guaranteed to be outside of the pruning circle of every other facility $f'$ and, therefore, is an RANN of $q$. Since $f$ is the nearest facility to $u$, $dist(u, f) \leq dist(u, f')$ for every other facility $f'$. Since $u$ lies outside $C_f$ (i.e., $dist(u, q) < x \times dist(u, f)$), we have $dist(u, q) < x \times dist(u, f')$. Thus, $u$ lies outside the pruning circle of $f'$. Hence if $u$ is not pruned by its nearest facility $f$ then it cannot be pruned by any other facility $f'$. □

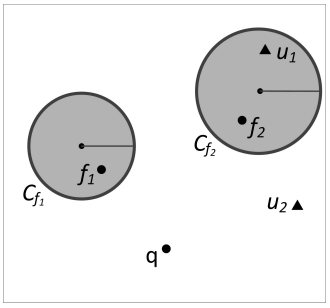



Fig. 10: RANN verification

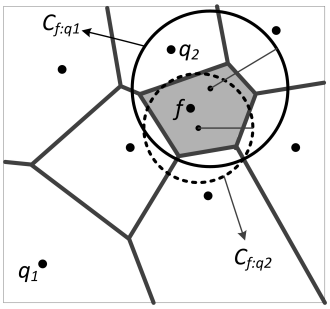



Fig. 11: Significant Facility

Now, consider a case where the system has $|Q|$ queries. A simple approach is to verify the user $u$ using the above lemma for each of the $|Q|$ queries. Assuming that the nearest facility $f$ is known, it requires $O(|Q|)$ to check which queries have $u$ as their RANNs. Next, we present an observation to reduce the number of queries that need to be considered to check which queries have $u$ as their RANNs.

First, we extend the notation to identify the pruning circle of a facility w.r.t. a query. Given a query $q$ and a facility $f$, we use $C_{f:q}$ to denote the pruning circle of $f$ with respect to query $q$. If the query is clear by context, we simply use $C_f$ (as we did earlier).

Assume that the Voronoi diagram of all facility points has been computed. Let $V_f$ denote the Voronoi cell of a facility $f$ (e.g., the shaded Voronoi cell in Fig. 11). A facility $f$ is called an insignificant facility for a query $q$ if $C_{f:q}$ completely contains its Voronoi cell $V_f$. On the other hand, a facility $f$ is called a significant facility for $q$ if $C_{f:q}$ does not completely contain $V_f$. In the example of Fig. 11, the facility $f$ is an insignificant facility for $q_1$ because the pruning circle $C_{f:q_1}$ (the solid circle) completely contains $V_f$. On the other hand, $f$ is a significant facility for $q_2$ because $C_{f:q_2}$ (the dotted circle) does not completely contain $V_f$. The next lemma shows that a user $u$ in a Voronoi cell $V_f$ can only be an RANN of a query for which $f$ is a significant facility.

**Lemma 6.** *Let $u$ be a user that lies in the Voronoi cell $V_f$ of a facility $f$. $u$ cannot be an RANN of any query $q$ for which $f$ is an insignificant facility.*

*Proof.* Since $u$ lies in the Voronoi cell $V_f$, $f$ is the nearest facility to $u$. Furthermore, since $f$ is an insignificant facility

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



for $q$ (i.e., $C_{f:q}$ completely contains $V_f$), $u$ is inside $C_{f:q}$. Therefore, $u$ cannot be an RANN of $q$ because it is contained in the pruning circle of its nearest facility (see Lemma 5). □

We use Lemma 5 and Lemma 6 to efficiently update RANNs of the queries. Specifically, for each Voronoi cell $V_f$, we create a list called $sigList$ that contains every query $q$ for which $f$ is a significant facility. A user $u$ that moves in a cell $V_f$ can be an RANN of only the queries in the $sigList$ of $V_f$. Hence, we only need $O|K|$ instead of $O|Q|$ to determine what queries have $u$ as its RANN where $|K|$ is the number of queries in $sigList$ of $V_f$. Our experiments show that $|K|$ is significantly smaller than $|Q|$ (around 1% of $|Q|$). In the example of Fig. 11, $sigList$ of $V_f$ contains only $q_2$ and a user lying in $V_f$ can be an RANN of only $q_2$.

To efficiently implement the above observations, we need techniques to efficiently determine the significant facilities for a given query $q$. Specifically, when a new query $q$ is registered at the system, the algorithm must be able to efficiently determine each of its significant facilities and add $q$ to its $sigList$. The next section provides the details on how to do this efficiently.

### 4.3 Efficiently identifying significant facilities

A naïve approach is to access each facility $f \in F$, construct its pruning circle $C_{f:q}$ and check whether it contains the Voronoi cell $V_f$ or not. However, this approach is not only computationally expensive but also incurs high I/O cost because the whole facility R*-tree (F-tree) needs to be accessed for each query. We observe that some nodes in the F-tree may not contain any significant facility and can be pruned. Therefore, an efficient approach is to iteratively access F-tree starting from the root node and accessing only the nodes that may contain some significant facilities. Before we present techniques to determine whether a node of F-tree may contain a significant facility or not, we first need to formalize how to check if a facility is significant for $q$ or not.

Consider the example of Fig. 12 where $V_f$ is represented as a set of vertices $v_1$ to $v_5$. The maximum distance between $f$ and $V_f$ is $dist(f, v_5)$ and the minimum distance between $f$ and its pruning circle $C_f$ is $dist(f, Z)$. Since $dist(f, Z) > dist(f, v_5)$, we can confirm that$C_f$ fully contains $V_f$. Therefore, to check if $f$ is a significant facility for $q$ or not (i.e., $C_f$ contains $V_f$ or not), we need to compute the maximum distance between $f$ and $V_f$ (denoted as $maxdist(f, V_f)$) and the minimum distance between $f$ and $C_f$ (denoted as $mindist(f, C_f)$). $maxdist(f, V_f)$ can be easily computed using $dist(f, v_i)$ for each vertex $v_i$ of the Voronoi cell $V_f$. Formally, $maxdist(f, V_f) = \max_{v_i \in V_f} dist(f, v_i)$. The next lemma shows that $mindist(f, C_f) = \frac{dist(q,f)}{x+1}$.

**Lemma 7.** *Given a query $q$, a multiplication factor $x > 1$, and a facility point $f$, $mindist(f, C_f) = \frac{dist(q,f)}{x+1}$.*

*Proof.* Consider the example of Fig. 12. Note that $mindist(f, C_f) = dist(f, Z) = dist(c, Z) - dist(c, f)$. Note that $dist(c, f) = dist(q, c) - dist(q, f)$. By definition of the pruning circle $C_f$ (see Definition 2), $dist(q, c) = \frac{x^2 \cdot dist(q,f)}{x^2-1}$. Thus, $dist(c, f) = \frac{x^2 \cdot dist(q,f)}{x^2-1} - dist(q, f) = \frac{dist(q,f)}{x^2-1}$. Since $dist(c, Z)$ is the radius of the pruning circle $C_f$, by definition of pruning circle, $dist(c, Z) = \frac{x \cdot dist(q,f)}{x^2-1}$. Therefore, $mindist(f, C_f) = \frac{x \cdot dist(q,f)}{x^2-1} - \frac{dist(q,f)}{x^2-1} = \frac{dist(q,f)}{x+1}$. □

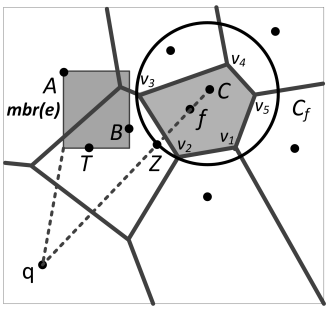



Fig. 12: Lemma 7

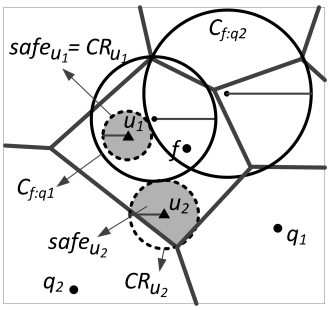



Fig. 13: safe zone

The next lemma summarizes the above observation. The proof is obvious and is omitted.

**Lemma 8.** *Given a query $q$ and a facility point $f$, $f$ is an insignificant facility of $q$ if $\frac{dist(q,f)}{x+1} > maxdist(f, V_f)$.*

Now, we are ready to extend the above lemma for a node of the facility R*-tree. Given a leaf or intermediate node $e$ of the facility R*-tree, we define $MaxMaxDist(e)$ as the maximum of $maxdist(f, V_f)$ for every facility $f \in e$, i.e., $MaxMaxDist(e) = \max_{f \in e} maxdist(f, V_f)$. Consider the node $e$ in Figure 12 (the shaded rectangle) which contains 3 facility points $A$, $B$ and $T$. Assuming that $maxdist(A, V_A)$, $maxdist(B, V_B)$ and $maxdist(T, V_T)$ are 2, 5 and 3, respectively, then $MaxMaxDist(e) = 5$.

The next lemma extends Lemma 8 for a node $e$ of the facility R*-tree.

**Lemma 9.** *A node $e$ of the facility R*-tree cannot contain any significant facility for a query $q$ if $\frac{mindist(q,e)}{x+1} > MaxMaxDist(e)$.*

*Proof.* We prove that every $f \in e$ is an insignificant facility for $q$. Since $dist(q, f) \geq mindist(q, e)$ and $maxdist(f, V_f) \leq MaxMaxDist(e)$, we have $\frac{dist(f,q)}{x+1} > maxdist(f, V_f)$. Therefore, $f$ is an insignificant facility (see Lemma 8). □

Note that the Voronoi diagram and $MaxMaxDist(e)$ are query independent and can be computed during preprocessing. Specifically, in the pre-processing phase, we first compute a Voronoi diagram of the facilities and calculate $maxdist(f, V_f)$ for each facility $f$. Finally, $MaxMaxDist(e)$ for each node $e$ in the facility R*-tree is computed in a bottom-up fashion and stored along with $e$.

### 4.4 Algorithms

In this section, we present our algorithms to continuously monitor the RANNs of queries. First, we show how to handle the case when a new query is issued (Section 4.4.1). Then, we present the algorithm to handle the case when a new user arrives (Section 4.4.2). In Section 4.4.3, we show how to handle the case when a query or a user is deleted. Finally, we explain how to update the result when one or more users change their locations (Section 4.4.4).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



Note that the initial results can be computed by first adding all the queries one by one (Section 4.4.1) and then adding each of the users (Section 4.4.2).

#### 4.4.1 Adding a query

When a new query $q$ is issued, we need to compute its initial results and to insert $q$ into the $sigList$ of each *significant facility* of $q$. Algorithm 3 provides the details. The algorithm initializes a list $L$ with the root of the facility R*-tree (F-tree) (line 1). The entries in the list $L$ are iteratively accessed and are processed as follows. If the accessed entry $e$ is an intermediate or leaf node and cannot be pruned using Lemma 9 (line 5), its children are inserted in the list $L$ (line 6).

**Algorithm 3 addQuery(q)**

```
1: insert root of F-tree in a list L
2: while L is not empty do
3:    remove an entry e
4:    if e is an intermediate or leaf node then
5:       if e is not pruned then                    ▷ apply Lemma 9
6:          insert all children of e in the list L
7:    else                                          ▷ e is a facility
8:       if C_{e:q} does not contain V_e then
9:          insert q in sigList of e
10:         for each user u in Voronoi cell of e do
11:            if u is outside C_{e:q} then
12:               insert u as RANN of q             ▷ Lemma 5
13:               insert q in qList of u
```

If $e$ is a facility point, we check whether it is a significant facility or not by checking if $V_e$ is contained by $C_e$ or not (line 8). If $e$ is a significant facility of $q$, $q$ is inserted in the *sigList* of the Vornoi cell of $e$ (line 9). Then, for each user $u$ in the Voronoi cell of $e$, the algorithm checks if $u$ is inside the pruning circle $C_{e:q}$ or not (line 10-12). If $u$ is outside $C_{e:q}$, it will be inserted as an RANN of $q$ and $q$ will be inserted in $qList$ of $u$ (line 13). $qList$ of a user $u$ stores all queries for which $u$ is an RANN. We need the $qList$ to enable us to efficiently find the queries for which $u$ is a result (see Section 4.4.3). The algorithm stops when the list $L$ becomes empty.

Note that at line 10 we need to obtain all users lying in a particular Voronoi cell. To do this, the server maintains the location of each user and the Voronoi cell in which it resides. For each Voronoi cell, the server also maintains a list of users residing in this cell. This list can be used to efficiently find all users in a particular Voronoi cell.

#### 4.4.2 Adding a user

When a new user $u$ arrives, we need to compute the safe zone of $u$ and update the result set of queries for which $u$ is an RANN. Algorithm 4 provides the details. The safe zone of $u$ is a region such that if $u$ is inside this region, the RANN of all queries in the system remain unchanged. Before we present the details of Algorithm 4, we provide an observation to construct the safe zone.

Recall that a user $u$ can be an RANN of only the queries in the *sigList* of $f$ where $f$ is the nearest facility of $u$ (Lemma 6). Furthermore, $u$ is a result of only a query $q_i \in sigList$ of $f$ for which $u$ is outside $C_{f:q_i}$ (Lemma 5). In other words, the user $u$ does not affect the results of any query $q_i \in sigList$ as long as it does not leave or enter the circle $C_{f:q_i}$. Therefore, we consider $C_{f:q_i}$ for every $q_i$ in *sigList* of $f$ and obtain the smallest circle such that as long as $u$ is inside it, it does not leave or enter any of $C_{f:q_i}$. Below, we formally describe this idea.

We use $mindist(u, C_{f:q_i})$ to denote the minimum distance between $u$ and a pruning circle $C_{f:q_i}$ where $f$ is the nearest facility to $u$. We compute $mindist(u, C_{f:q_i})$ for every query $q_i$ in the $sigList$ of $f$. The minimum value of $mindist(u, C_{f:q_i})$ is maintained and is stored as $r_{\mathcal{C}}$, i.e $r_{\mathcal{C}} = \min_{q_i \in sigList_f} mindist(u, C_{f:q_i})$. We create a *critical circle* of $u$ (denoted as $CR_u$) centered at $u$ with radius $r_{\mathcal{C}}$. Consider the example of user $u_1$ in Fig. 13 that lies in $V_f$. The $sigList$ contains $q_1$ and $q_2$ and the pruning circles $C_{f:q_1}$ and $C_{f:q_2}$ are also shown. Since $\min_{q_i \in sigList_f} mindist(u, C_{f:q_i}) = mindist(u_1, C_{f:q_2})$, $r_{\mathcal{C}} = mindist(u_1, C_{f:q_2})$. The shaded circle $CR_{u_1}$ is the *critical circle* for $u_1$. Note that as long as $u_1$ is in this circle, it does not enter or leave any pruning circle and, therefore, does not affect the result of $q_1$ or $q_2$. Fig. 13 also shows the $CR_{u_2}$ for the user $u_2$.

Note that the construction of $CR_u$ only considers the queries in the $sigList$ of facility $f$ where $f$ is the nearest facility to $u$. Consequently, $CR_u$ is valid as long as $u$ is inside the Voronoi cell $V_f$ of $f$. If $u$ leaves the Voronoi cell of $f$, the query results may change. Consider an example of $u_2$ in Fig. 13. If $u_2$ leaves the Voronoi cell, even though it may still be inside its critical circle $CR_{u_2}$, the query results may change. Hence, we construct the safe zone of $u$ (denoted as $safe_u$) as the intersection region of $CR_u$ and the Voronoi cell $V_f$ of $f$, i.e., $safe_u = CR_u \cap V_f$. In Fig 13, the safe zones of $u_1$ and $u_2$ are the shaded regions.

Now, we are ready to present our algorithm to handle a new user $u$ that arrives. First, Algorithm 4 issues a nearest neighbour query on facility R*-tree to find its nearest facility $f$ (line 1). Then, the Voronoi cell $V_f$ of $f$ is obtained from the pre-computed Voronoi diagram. Subsequently, it accesses the queries in $sigList$ of $V_f$ iteratively (line 3). For each query $q_i$ in the $sigList$ of $f$, the algorithm checks whether $u$ is inside $C_{f:q_i}$ or not. If $u$ is outside $C_{f:q_i}$, it is inserted as RANN of $q_i$ and $q_i$ is inserted in $qList$ of $u$ (line 4-6). During the iteration, the algorithm maintains $r_{\mathcal{C}}$ (line 7-8) which implies that $r_{\mathcal{C}}$ corresponds to the radius of the critical circle when every $q_i$ has been accessed. The safe zone of $u$ ($safe_u$) is the intersection of $CR_u$ and the Voronoi cell of $f$ (line 10).

**Algorithm 4 addUser(u)**

```
1: get the nearest facility f of user u
2: r_C ← ∞
3: for each q_i ∈ sigList of V_f do
4:    if u is outside C_{f:q_i} then
5:       insert u as RANN of q_i
6:       insert q_i in qList of u
7:    if r_C < mindist(u, C_{f:q_i}) then
8:       r_C ← mindist(u, C_{f:q_i})
9: create critical circle CR_u centered at u with radius r_C
10: safe_u ← CR_u ∩ V_f
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



#### 4.4.3 Deleting a query or a user

Recall that $qList$ of a user $u$ contains every query $q$ for which $u$ is an RANN. When a user $u$ is deleted, we delete $u$ from the result set of every query in its $qList$ and the affected queries are notified of the updated results.

When a query $q$ is deleted, we retrieve every facility $f$ for which $q$ is in its *sigList* and delete $q$ from the $sigList$. Note that for each query, we can easily maintain a list of its significant facilities, e.g., when a significant facility is identified at line 9 of Algorithm 3. Then, we remove $q$ from the $qList$ of every user $u$ that has $q$ in its $qList$ (i.e., $u$ is an RANN of $q$). Note that when a query is removed, the safe zone of some users may become larger. However, we decide not to update the safe zone of such users because it may incur unnecessary computation and communication cost becasue the new safe zones need to be computed and sent to the users. The safe zone of those users will be updated when they leave their safe zones. This does not affect the correctness of the algorithm.

#### 4.4.4 Handling the movement of users

Recall that a user's movement does not affect the results of any query as long as the user remains in its respective safe zone. Therefore, the user sends its location to the server only when it leaves its safe zone. In this case, the results can be easily maintained by first deleting the user from the system and then adding the user (as described in the previous sections). However, some simple yet effective optimizations are possible as described below.

Note that Algorithm 4 (where we add a user $u$) requires the nearest facility to $u$ (line 1). To find it, a simple approach is to issue a nearest neighbour query on the facility R*-tree for every location update by the user. However, it may incur unnecessary I/O cost. To improve this, we first check if $u$ is still inside the same Voronoi cell. If it is, we do not need to compute its nearest facility because the nearest facility and the Voronoi cell that contains $u$ remain unchanged. Otherwise, we handle the update as follows.

Let $f_{old}$ (resp. $V_{f_{old}}$) be the previous nearest facility (resp. Voronoi cell) of $u$ and $f_{new}$ (resp. $V_{f_{new}}$) is the current nearest facility (resp. Voronoi cell) of $u$. If $u$ is outside $V_{f_{old}}$, we need to find the current nearest facility to $u$ ($f_{new}$). Note that $dist(u, f_{new}) < dist(u, f_{old})$. Thus, a nearest neighbour query is issued with a simple modification that every entry $e$ in the facility R*-tree is ignored if $mindist(u, e) > dist(u, f_{old})$.

## 5 Experiments

All algorithms were implemented in C++ and experiments were run on Intel Core *I*5 2.3GHz PC with 8GB memory running on Debian Linux. We present the details of the experimental setup and results for snapshot RANN queries and continuous RANN queries in Section 5.1 and Section 5.2, respectively.

### 5.1 Snapshot RANN Queries

#### 5.1.1 Evaluated Algorithms

To the best of our knowledge, there is no prior algorithm to solve RANN queries and extending existing algorithms is non-trivial (see Section 3.2.1). We compare the following algorithms in this paper.

*1. Improved Range Query (IRQ).* A naïve approach to answer a snapshot RANN query is to issue a boolean range query for each user $u$ to verify whether it is an RANN or not (see the verification phase in Section 3.3). However, this approach is too expensive. In an earlier version [14] of this paper, we proposed a significantly improved approach, called improved range query (IRQ), that provides an improvement of up to two orders of magnitude. The main idea is to extend the boolean range queries for intermediate nodes of the user R*-tree and prune the intermediate nodes that cannot contain any RANN (see [14] for details).

*2. Voronoi.* Our techniques to handle continuous RANN queries include Algorithm 3 that adds a new query $q$ to the system. This algorithm can be immediately applied to compute the snapshot RANN queries. Recall that the algorithm relies on a pre-computed Voronoi diagram and $MaxMaxDist(e)$ for each node $e$ of R*-tree. Therefore, it is denoted as **Voronoi** in this section. We also pre-compute and store, for each Voronoi cell $V$, the users lying in $V$. This allows the algorithm to efficiently access the candidate RANNs (see line 10 in Algorithm 3). Note that the pre-computation does not depend on query parameters (e.g., query location or $x$-factor).

*3. Standard.* In contrast to *Voronoi* that requires a pre-computed Voronoi diagram and modifications (e.g., $MaxMaxDist(e)$) to the standard R*-tree, our algorithm presented in Section 3 can be applied on any standard branch-and-bound indexes such as R*-tree and Quadtree etc. Therefore, we call it **Standard** in this paper to emphasize that this algorithm can be applied on existing standard indexes without requiring any changes to the indexes.

#### 5.1.2 Experimental Settings

Experimental settings are similar to a recent experimental study [6] on R$k$NN queries. We use real data sets containing $175,812$ points from North America (called NA data set hereafter) and 2.6 million points from Los Angeles (LA). We also generate several synthetic data sets containing $1,000$ to 1 million points following normal distributions. Unless mentioned otherwise, each data set is randomly divided into two sets of almost equal size, one corresponding to the facilities and the other to the users.

We randomly select $2,000$ points from the facility data set and treat them as query points. The results report the average CPU cost and I/O cost for a single RANN query. The I/O cost for IRQ and *Standard* is the cost to access the facility R*-tree and the user R*-tree while the I/O cost for *Voronoi* includes the cost to access the modified facility R*-tree as well as accessing the lists of users lying in particular Voronoi cells accessed by the algorithm. The page size is set to $4,096$ Bytes.

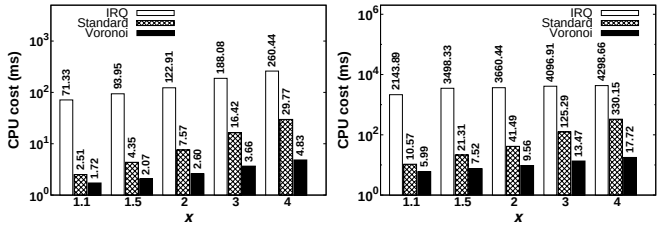
#### 5.1.3 Results

**Effect of the $x$ factor**. Fig. 14 shows the effect of the $x$ factor on the CPU cost of the three algorithms on NA and LA data sets. Both of our algorithms are one to two orders of magnitude faster than IRQ. Furthermore, *Voronoi* is up to 20 times more efficient than *Standard* and scales better as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



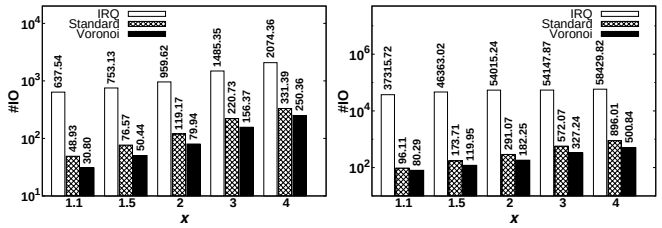
the value of $x$ increases (note that Fig. 14 uses log scale on y-axis).




(a) NA data set (b) LA data set

Fig. 14: Effect of the $x$ factor on CPU cost

Fig. 15 shows the effect of the $x$ factor on the I/O cost of the three algorithms on NA and LA data sets. The I/O cost of both of our algorithms is one to two orders of magnitude smaller than the I/O cost of IRQ (note that log scale is used on y-axis). The I/O cost of *Voronoi* is 60% to 80% of the I/O cost of *Standard* on both data sets. Since both of our algorithms are more than an order of magnitude better than IRQ, in the rest of this section, we omit IRQ to better illustrate the comparison between our algorithms.

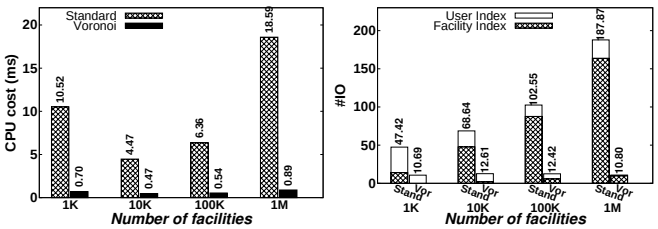



(a) NA data set (b) LA data set

Fig. 15: Effect of the $x$ factor on I/O cost

**Effect of number of facilities**. In Fig. 16, we vary the number of facilities from 1000 to 1 million and the number of users is fixed to $100K$. The sets of facilities and users are generated following normal distribution. Fig. 16(a) shows the CPU cost of both algorithms. *Voronoi* significantly outperforms *Standard* and is less affected by the change in the number of facilities. Note that the cost of both algorithms is larger if the number of facilities is too small or too large. The reason is as follows. When the number of facilities is too small (e.g., 1000), the total area that can be pruned is smaller due to the lower density of the facilities. This results in a larger number of candidates and RANNs which increases the overall CPU cost. On the other hand, when the number of facilities is too large (e.g., 1 million), the cost of the pruning phase increases significantly. This is because the algorithm needs to access a larger number of R*-tree entries to prune the search space.

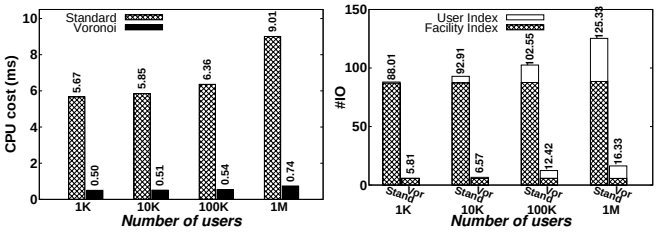
Fig. 16(b) studies the effect of the number of facilities on the I/O cost of both algorithms. The total I/O cost is broken down into the number of I/Os on the facility index (i.e., facility R*-tree) and user index (i.e., user R*-tree in *Standard* and the lists of users stored for each Voronoi cell in *Voronoi*). As the number of facilities increases, the size of facility index increases which results in a larger number of I/Os on the facility index as shown in Fig. 16(b). The




(a) CPU cost (b) I/O cost

Fig. 16: Effect of the number of facilities ($100K$ users)

I/O cost on the user index decreases for both algorithms as the number of facilities increases. This is because, as the number of facilities increases, a larger area can be pruned which results in pruning more entries of the user index.




(a) CPU cost (b) I/O cost

Fig. 17: Effect the number of users ($100K$ facilities)

**Effect of number of users.**. In Fig. 17, we vary the number of users from 1000 to 1 million and fix the number of facilities to $100K$. Fig. 17(a) and Fig. 17(b) show that the CPU cost and I/O cost of both algorithms increase as the number of users increases. This is because the number of candidate users and RANNs increases as the number of users increases which results in a higher cost to filter and verify the users. *Voronoi* significantly outperforms *Standard* across all data sets. The above experiments show that *Voronoi* must be preferred if the underlying indexes can be changed. However, *Standard* is a good choice if the system owner does not want to modify the underlying standard indexes.

**Comparing different pruning/filtering strategies.** In Section 3.3, we briefly discussed aggressive pruning approach and moderate pruning approach. We also designed another strategy called *aggressive filtering* which uses the moderate pruning approach but filtering is done aggressively. Specifically, in our main approach, if a user entry $U$ is not filtered by the shortlisted facilities, all its children are considered to be candidates which need to be verified. In the *aggressive* filtering approach, if a user entry $U$ is not filtered by the shortlisted facilities, we traverse facility R*-tree to see if this entry can be pruned by considering other facilities in the facility R*-tree. This is done using the same idea used in the improved range query algorithm discussed in Section 5.1 of the conference version [14] of this paper. In this section, we compare the three different strategies.

In Fig. 18, we compare aggressive pruning approach (shown as aggPruning), aggressive filtering approach (shown as aggFiltering) and moderate pruning approach by varying the number of facilities. Moderate pruning approach outperforms both aggressive pruning and aggressive

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



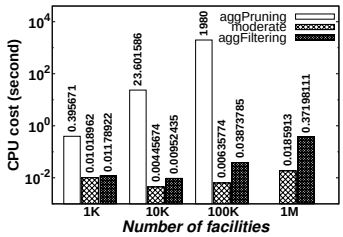



Fig. 18: Comparing different pruning/filtering strategies

filtering approaches – the cost of aggPruning is not shown for 1 Million facilities because it failed to return results even after a couple of days. The aggressive pruning approach aims to maximize the pruned space and ends up shortlisting a much higher number of facilities which results in a much higher pruning/filtering cost. On the other hand, it does not significantly reduce the number of candidates. A detailed explanation is provided in Appendix.

Note that the aggressive filtering (shown as aggFiltering) is significantly more effective than the aggressive pruning approach but it is still worse than moderate approach. However, we remark that, in the data sets that may exhibit worst-case scenarios where most of users are failed to be pruned by the facilities shortlisted by moderate pruning approach, the aggressive filtering approach may be the right choice. Therefore, the applications where avoiding the worst-case scenarios is important, the aggressive filtering approach may be preferable.

## 5.2 Continuous RANN Queries

### 5.2.1 Competitor

To the best of our knowledge, there is no existing algorithm for continuously monitoring RANN queries. Influence zone [8] is the *state-of-the-art* algorithm for continuous monitoring of R$k$NN queries. We extend the techniques proposed in [8] to continuously monitor RANN queries by extending the notion of influence zone for RANN queries, i.e., an area such that a user is an RANN of a query if and only if the user is inside this area.

Recall that a user $u$ is an RANN of a query $q$ if and only if it lies outside the pruning circle of every facility (Lemma 1 and Lemma 2). Thus, the influence zone can be defined as the area outside the pruning circles of all facilities. For example, in Fig 5, influence zone is the white area and a user $u$ can be an RANN of $q$ if and only if $u$ lies in the white area. Thus, a straightforward approach to compute influence zone of a query $q$ is to consider the pruning circles for every facility. Influence zone of each query $q \in Q$ can be then computed and indexed using a grid (similar to [8]) and the RANNs of all queries can be monitored using the ideas presented in [8] (see Section 2.2).

However, note that the above approach requires computing $|Q| \times |F|$ pruning circles where $|Q|$ and $|F|$ denote the total number of queries and facilities, respectively. This is not only computationally expensive but also requires huge memory to index all circles in the grid. As explained in Section 3.2.1, it is not trivial to reduce the number of pruning circles because unlike R$k$NN queries, the users that are very far may still be the RANNs. Nevertheless, to optimize the performance, we carefully design a pruning technique that significantly reduces the number of pruning circles (see Appendix). Our experiments show that this reduces the total number of pruning circles by 30% to 65%.

We partition the data space into an $N \times N$ grid structure ($N$ is set to 64 in the experiments as this gives the best overall performance). For each grid cell, we maintain two lists: *q-list* and *c-list*. The *q-list* of a cell $c$ contains queries whose pruning circles do not overlap $c$. When a user moves into $c$, it will be immediately inserted as an RANN of each query in the *q-list* of $c$. *c-list* of a cell $c$ stores each query $q$ for which there exist at least one facility whose pruning circle with respect to $q$ overlaps $c$. For each $q$ in *c-list* of $c$, a list $l_{q:c}$ containing facilities whose pruning circle overlaps $c$ is maintained. When a user $u$ moves into $c$, it is checked against each query $q$ in *c-list* of $c$. If $u$ is outside the pruning circles of each facility in $l_{q:c}$, it is inserted as an RANN of $q$.

We also assign a safe zone for each user. Similar to the Voronoi-based algorithm, for each user $o$ in cell $c$, we iterate over all pruning circles in *c-list* of $c$ to get the minimum distance between $o$ and circles in *c-list* of $c$. We set this distance as the radius of safe circle. The safe zone of $o$ is the intersection of the safe circle and the cell $c$.

### 5.2.2 Experimental Settings

We compare our algorithm *Voronoi* with the extended influence zone algorithm (denoted as InfZone). We use a real world data set containing point of interests from Los Angeles (LA). The moving objects (i.e., users) are generated by simulating moving cars on the road network of LA using the well-known Brinkhoff data generator [31]. The parameters used in the experiments are shown in Table 1 and the default values are shown in bold.

TABLE 1: Experiments Parameters

| Parameters | Range |
|---|---|
| $x$ | 1.1, **1.5**, 2, 3, 4 |
| Number of facilities (X 1000) | $10, 50,$ **100**, $150$ |
| Number of users (X 1000) | $10, 50,$ **100**, $150$ |
| Users' speed (Km/hr) | 40, 60, **80**, 100, 120 |
| Users' Mobility (%) | 20, 40, 60, 80, **100** |

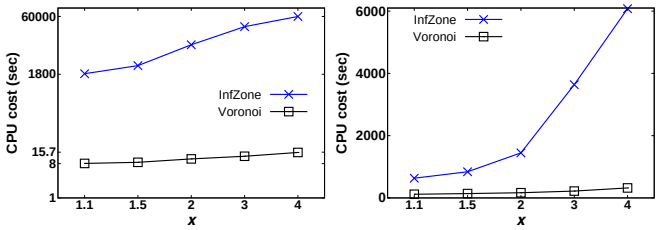
Due to its high memory usage, InfZone cannot handle more than 1000 continuous RANN queries for all data settings. Therefore we use 1000 continuous queries as default. Each query is a randomly selected point from the facility data set and we monitor all 1000 queries for 100 timestamps. We report the total initial cost and the total monitoring cost. The total initial cost is the cost to compute the initial results of all queries. The total monitoring cost is the cost to continuously update the results of the affected queries for 100 timestamps.

### 5.2.3 Results

**Effect of the $x$ factor.** Fig. 19 studies the effect of the $x$ factor on both algorithms. As expected, the cost of each algorithm increases with the increase of $x$ factor because a smaller area is pruned when $x$ is larger. Fig. 19(a) shows that the initial computation cost of *Voronoi* is two to three orders of magnitude lower than that of InfZone and *Voronoi* scales much better (note that log scale is used on y-axis). Fig. 19(b) shows that *Voronoi* outperforms InfZone by up to two orders

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2766065, IEEE Transactions on Knowledge and Data Engineering



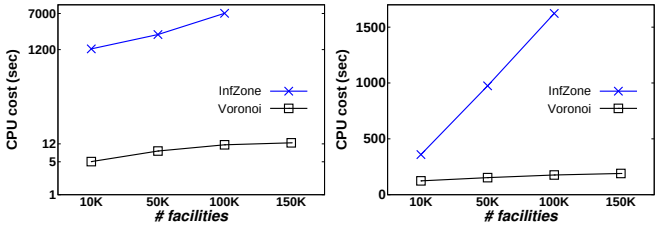



(a) Initial cost (b) Monitoring cost

Fig. 19: Effect of the $x$ factor

of magnitude and scales much better as the value of $x$ increases. The initial computation cost of *Voronoi* is quite low as shown in the previous section for the snapshot RANN queries. The continuous monitoring cost is also very small due to the effective use of the Voronoi cells and significant facilities. In contrast, InfZone is significantly more expensive mainly because it requires computing and indexing a large number of pruning circles for each query.

**Effect of number of facilities.** In Fig. 20, we study the effect of the number of facilities on both algorithms. InfZone failed to run for $150,000$ facilities because it ran out of memory. The cost of both algorithms increases with the increase in number of facilities mainly because the number of significant facilities and the number of pruning circles increase as the number of facilities increases. Fig. 20 shows that *Voronoi* significantly outperforms InfZone in terms of both the initial computation cost and the continuous monitoring cost and scales better. In the rest of the experiments, we only compare the monitoring cost of the two algorithms because the initial cost of InfZone is two to three orders of magnitude higher than *Voronoi* for all data settings.




(a) Initial cost (b) Monitoring cost

Fig. 20: Effect of number of facilities

**Effect of number of users.** Fig. 21 shows the effect of number of users on the monitoring cost of both algorithms. As expected, the monitoring cost of both algorithms increases with the increase in number of users. *Voronoi* significantly outperforms InfZone and scales much better.

**Effect of mobility.** Fig. 22 studies the effect of mobility which correspond to percentage of the users that move between two timestamps, e.g., $80\%$ mobility corresponds to the data set where 80% of the total users change their locations between two timestamps and the rest of the users are static (e.g., car waiting on traffic light). As expected, the monitoring cost of both algorithms increases with the increase in mobility. *Voronoi* significantly outperforms InfZone and scales much better which is mainly because the safe zones created by *Voronoi* are larger and it takes longer

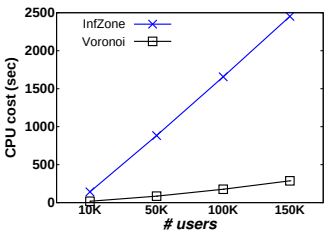



Fig. 21: Effect of # of users

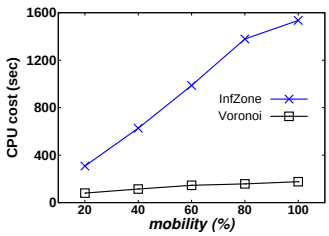
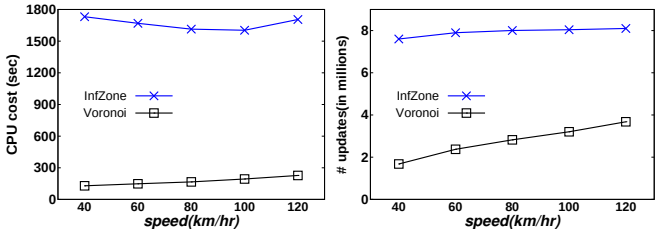



Fig. 22: Effect of mobility

for a user to leave its safe zone which results in requiring fewer updates.




(a) Monitoring cost (b) Number of updates

Fig. 23: Effect of the user's speed

**Effect of the speed.** Fig. 23 studies the effect of users' speed on the monitoring cost of both algorithms. Fig. 23(a) shows that *Voronoi* significantly outperforms InfZone. Fig. 23(b) shows the total number of updates in 100 timestamps where an update corresponds to the instance when a user leaves its safe zones. The monitoring cost of *Voronoi* increases with the increase in speed because the number of users that leave their respective safe zones increases with the increase in the speed. Although the number of updates for InfZone also increases with the increase in speed, its monitoring cost is relatively stable. This is because the initial positions of the users generated by Brinkhoff data generator are randomly chosen. Therefore, the initial positions of the users are different in each data set and the trend is difficult to predict because a data sets where more users are located in dense areas will have higher costs.

## 6 Conclusion

In this paper, we propose a variant of RNN queries called reverse approximate nearest neighbors (RANN) queries. An RANN query relaxes the definition of influence using the relative distances between the users and the facilities. RANN queries are motivated by our observation that R$k$NN queries may be unable to properly capture the notion of influence. Based on non-trivial observations, we propose efficient algorithms for snapshot and continuous RANN queries. Our extensive experimental study on real and synthetic data sets demonstrate that our algorithms significantly outperform the competitors.

## References

[1] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," in *SIGMOD*, 2000, pp. 201–212.

[2] I. Stanoi, D. Agrawal, and A. E. Abbadi, "Reverse nearest neighbor queries for dynamic databases," in *ACM SIGMOD Workshop*, 2000.

[3] M. A. Cheema, X. Lin, W. Wang, W. Zhang, and J. Pei, "Probabilistic reverse nearest neighbor queries on uncertain data," *IEEE Trans. Knowl. Data Eng.*, 2010.

[4] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi, "Discovery of influence sets in frequently updated databases," *PVLDB*, 2001.

[5] Y. Tao, D. Papadias, and X. Lian, "Reverse knn search in arbitrary dimensionality," *PVLDB*, pp. 744–755, 2004.

[6] S. Yang, M. A. Cheema, X. Lin, and W. Wang, "Reverse k nearest neighbors query processing: Experiments and analysis," *PVLDB*, 2015.

[7] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan, "FINCH: Evaluating reverse k-nearest-neighbor queries on location data," *PVLDB*, 2008.

[8] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang, "Influence zone: Efficiently processing reverse k nearest neighbors queries," in *ICDE*, 2011, pp. 577–588.

[9] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li, "Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks," *VLDB J.*, pp. 69–95, 2012.

[10] S. Yang, M. A. Cheema, X. Lin, and Y. Zhang, "SLICE: Reviving regions-based pruning for reverse k nearest neighbors queries," in *ICDE*, 2014, pp. 760–771.

[11] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg, "Reverse top-k queries," in *ICDE*, 2010, pp. 365–376.

[12] M. A. Cheema, Z. Shen, X. Lin, and W. Zhang, "A unified framework for efficiently processing ranking related queries," in *EDBT*, 2014.

[13] S. Yang, M. A. Cheema, X. Lin, Y. Zhang, and W. Zhang, "Reverse k nearest neighbors queries and spatial reverse top-k queries," in *VLDB Journal*, 2016.

[14] A. Hidayat, M. A. Cheema, and D. Taniar, "Relaxed reverse nearest neighbors queries," in *SSTD*, 2015.

[15] T. Emrich, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle, "Incremental reverse nearest neighbor ranking in vector spaces," in *SSTD*, 2009.

[16] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun, "High dimensional reverse nearest neighbor queries," in *CIKM*, 2003.

[17] E. Achtert, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle, "Reverse k-nearest neighbor search in dynamic and general metric databases," in *EDBT*, 2009, pp. 886–897.

[18] M. Sharifzadeh and C. Shahabi, "Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries," *PVLDB*, vol. 3, no. 1, pp. 1231–1242, 2010.

[19] M. A. Cheema, X. Lin, Y. Zhang, W. Wang, and W. Zhang, "Lazy updates: An efficient technique to continuously monitoring reverse knn," *PVLDB*, pp. 1138–1149, 2009.

[20] T. Bernecker, T. Emrich, H.-P. Kriegel, M. Renz, , and S. Z. A. Züfle, "Efficient probabilistic reverse nearest neighbor query processing on uncertain data," *PVLDB*, pp. 669–680, 2011.

[21] Z. Xu and H. Jacobsen, "Adaptive location constraint processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, 2007, pp. 581–592. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247545

[22] Y. Lu, J. Lu, G. Cong, W. Wu, and C. Shahabi, "Efficient algorithms and cost models for reverse spatial-keyword *k*-nearest neighbor search," *ACM Trans. Database Syst.*, vol. 39, no. 2, pp. 13:1–13:46, 2014. [Online]. Available: http://doi.acm.org/10.1145/2576232

[23] S. Wang, M. A. Cheema, X. Lin, Y. Zhang, and D. Liu, "Efficiently computing reverse k furthest neighbors," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, 2016, pp. 1110–1121. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498317

[24] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," in *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 2007, pp. 291–302. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p291-dellis.pdf

[25] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan, "Continuous reverse k-nearest-neighbor monitoring," in *MDM*, 2008, pp. 132–139.

[26] M. A. Cheema, W. Zhang, X. Lin, and Y. Zhang, "Efficiently processing snapshot and continuous reverse k nearest neighbors queries," *VLDB Journal*, 2012.

[27] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest neighbor and reverse nearest neighbor queries for moving objects," in *IDEAS*, 2002, pp. 44–53.

[28] T. Xia and D. Zhang, "Continuous reverse nearest neighbor monitoring," in *ICDE*, 2006, pp. 77–86.

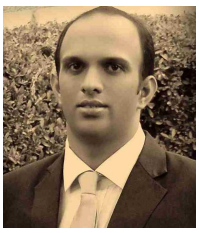
[29] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang, "Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors," in *ICDE*, 2007, pp. 806–815.

[30] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990.

[31] T. Brinkhoff and O. Str, "A framework for generating network-based moving objects," *Geoinformatica*, vol. 6, p. 2002, 2002.


**Arif Hidayat** is currently a PhD student in the Faculty of Information Technology, Monash University, Australia. He received his Master of Applied Information Technology from Monash University in 2009. He completed his Bachelor of Science in Agro-Industrial Technology from Brawijaya University, Indonesia in 2004. His current research interests include spatial databases and location-based services.

**Shiyu Yang** received the BS and MS degrees from the Dalian University of Technology, China, and the PhD degree from the University of New South Wales, Australia. He is currently a postdoc research fellow in the School of Computer Science and Engineering, University of New South Wales, Australia. His research interests include spatial databases and location-based services. He has published papers in conferences and journals including ICDE, PVLDB, VLDB Journal and the Computer Journal.



**Muhammad Aamir Cheema** is a Senior Lecturer at Clayton School of Information Technology, Monash University, Australia. He obtained his PhD from UNSW Australia in 2011. He is the recipient of 2012 Malcolm Chaikin Prize for Research Excellence in Engineering, 2013 Discovery Early Career Researcher Award, 2014 Dean's Award for Excellence in Research by an Early Career Researcher. His PhD thesis was nominated for SIGMOD Jim Gray Doctoral Dissertation Award and ACM Doctoral Dissertation Competition. He has won two CiSRA best research paper awards (in 2009 and 2010), two invited papers in the special issue of IEEE TKDE on the best papers of ICDE (2010 and 2012), and two best paper awards at WISE 2013 and ADC 2010, respectively. He served as PC co-chair for ADC 2015, ADC 2016, 8th ACM SIGSPATIAL Workshop ISA 2016, WWW International Workshop on Social Computing 2017, proceedings chair for DASFAA 2015, tutorial co-chair for APWeb 2017 and publicity co-chair for ACM SIGSPATIAL 2017.

**David Taniar** holds Bachelor, Master, and PhD degrees - all in Computer Science, with a particular specialty in Databases. His current research interests cover spatial query processing, and parallel databases. He has published a book: High Performance Parallel Database Processing and Grid Databases (John Wiley & Sons, 2008). He has published over 130 research papers that can viewed at the DBLP server. He is a founding editor-in-chief of International Journal of Data Warehousing and Mining, International Journal of Web and Grid Services, and International Journal of Web Information Systems. He is currently an Associate Professor at the Faculty of Information Technology, Monash University, Australia.